4 Experiment 4 Interrupt and I/O Interfacing

This experiment further consolidates the programmer's view of computer architecture. It does this by giving you details of the AVR processor's interrupt modes and I/O interfacing. This experiment also shows how you can interface to input/output devices, using interrupts.

4.1 Aim

This experiment aims to:

- Teach you all interrupts supported by AVR.
- Gain experience with interrupt-based AVR assembly language programming.
- Give details of external interrupts and how to service them.
- Show the support that the AVR instruction set architecture has for interfacing to input/output devices.
- Demonstrate simple and more complex peripherals, as well as the use of timers, keypad, and LCD display in the microcontroller-based systems.

4.2 Preparation

It is important that you prepare for each laboratory experiment, so that you can use your time (and your partner's time) most effectively. For this particular, experiment, you should do the following before coming in to the Laboratory:

- Read through this experiment in detail, trying to understand what you will be doing.
- Skim-read the section on Programming Style in Experiment 1 and 2.
- Quickly read through the relevant material from your lecture notes for this course.

It is highly recommended that you:

- Type up or modify the necessary files in this experiment, to save time in class, and
- Run through the experiment at home using AVR Studio and AVR Microcontroller Board.

4.3 Save Your Work

The computer in the lab are also used by other students in this course. So when you finish your lab, please save all your work to your floppy disk and DELETE all files associated with your lab if you do not want other students to use your files.

4.4 Part 1: Interrupt

Very often we have to react conditions or other events such as change on an input pin. You can program such a reaction by writing a loop, asking whether a change on the pin has occurred. This method is called polling. If there are no other things to do and reaction time does not matter, you can do this with the processor. Otherwise, you need to program an *interrupt*.

In general, an interrupt is a signal generated by an I/O device, or a timer (called hardware interrupt). AVR does not support any instruction to generate a software interrupt. However, programmers can use external interrupts as software interrupts. An interrupt is not handled by CPU unless the I bit in Program Status Register is set. The relevant instruction that does this is `sei`. The AVR provides several different interrupt sources. These interrupts and the separate Reset Vector each have a separate program vector in the program memory space. All interrupts are assigned individual enable bits which must be written logic one together with the Global Interrupt Enable bit in the Status Register in order to enable the interrupt.

The lowest addresses in the program memory space are by default defined as the Reset and Interrupt Vectors. The complete list of vectors is shown on the next page.

The list also determines the priority levels of the different interrupts. The lower the address the higher is the priority level. RESET has the highest priority, and next is INT0, the External Interrupt Request 0.

When an interrupt occurs, the Global Interrupt Enable I-bit is cleared and all interrupts are disabled. The user software can write logic one to the I-bit to enable nested interrupts. All enabled interrupts can then interrupt the current interrupt routine. The I-bit is automatically set when a Return from Interrupt instruction `reti` is executed.

There are basically two types of interrupts. The first type is triggered by an event that sets the interrupt flag. For these interrupts, the Program Counter is vectored to the actual Interrupt Vector in order to execute the interrupt handling routine, and hardware clears the corresponding interrupt flag. Interrupt flags can also be cleared by writing a logic one to the flag bit position(s) to be cleared. If an interrupt condition occurs while the corresponding interrupt enable bit is cleared, the interrupt flag will be set and remembered until the interrupt is enabled, or the flag is cleared by software. Similarly, if one or more interrupt conditions occur while the Global Interrupt Enable bit is cleared, the corresponding interrupt flag(s) will be set and remembered until the Global Interrupt Enable bit is set, and will then be executed by order of priority.

The second type of interrupts will trigger as long as the interrupt condition is present. These interrupts do not necessarily have interrupt flags. If the interrupt condition disappears before the interrupt is enabled, the interrupt will not be triggered. When the AVR exits from an interrupt, it will always return to the main program and execute one more instruction before any pending interrupt is served. Note that the Status Register is not automatically stored when entering an interrupt routine, nor restored when returning from an interrupt routine. This must be handled by software by `push` onto or `pop` from the stack. When using the `cli` instruction to disable interrupts, the interrupts will be immediately disabled. No interrupt will be executed after the `cli` instruction, even if it occurs simultaneously with the `cli` instruction. When using the `sei` instruction to enable interrupts, the instruction following `sei` will be executed before any pending interrupts.

The most typical and general program setup for the Reset and Interrupt Vector Addresses in ATmega64 is shown in Figure 1.

Vector No.	Program Address	Source	Interrupt Definition
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	0x0002	INT0	External Interupt Request 0
3	0x0004	INT1	External Interupt Request 1
4	0x0006	INT2	External Interupt Request 2
5	0x0008	INT3	External Interupt Request 3
6	0x000A	INT4	External Interupt Request 4
7	0x000C	INT5	External Interupt Request 5
8	0x000E	INT6	External Interupt Request 6
9	0x0010	INT7	External Interupt Request 7
10	0x0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	0X0014	TIMER2 OVF	Timer/Counter2 Overflow
12	0X0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	0x0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	0x001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	0x001C	TIMER1 OVF	Timer/Counter0 Overflow
16	0x001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
18	0x0022	SPI, STC	SPI Serial Transfer Complete
19	0x0024	USART0, RX	USART0, Rx Complete
20	0x0026	USART0, UDRE	USART0 Data Register Empty
21	0x0028	USART0, TX	USART0, Tx Complete
22	0x002A	ADC	ADC Conversion Complete
23	0x002C	EE READY	EEPROM Ready
24	0x002E	ANALOG COMP	Analog Comparator
25	0x0030	TIMER1 COMPC	Timer/Counter1 Cmpare Match C
26	0x0032	TIMER3 CAPT	Timer/Counter3 Capture Event
27	0x0034	TIMER3 COMPA	Timer/Counter3 Compare Match A
28	0x0036	TIMER3 COMPB	Timer/Counter3 Compare Match B
29	0x0038	TIMER3 COMPC	Timer/Counter3 Compare Match C
30	0x003A	TIMER3 OVF	Timer/Counter3 Overflow
31	0x003C	USART1, RX	USART1, Rx Complete
32	0x003E	USART1, UDRE	USART1 Data Register Empty
33	0x0040	USART1, TX	USART1, Tx Complete
34	0x0042	TWI	Two-wire Serial Interface
35	0x0044	SPM READY	Store Program Memory Ready

```
0x0000    rjmp       RESET       ; Reset Handler
0x0002    rjmp       EXT_INT0    ; IRQ0 Handler
0x0004    rjmp       EXT_INT1    ; IRQ1 Handler
0x0006    rjmp       EXT_INT2    ; IRQ2 Handler
0x0008    rjmp       EXT_INT3    ; IRQ3 Handler
0x000A    rjmp       EXT_INT4    ; IRQ4 Handler
0x000C    rjmp       EXT_INT5    ; IRQ5 Handler
0x000E    rjmp       EXT_INT6    ; IRQ6 Handler
0x0010    rjmp       EXT_INT7    ; IRQ7 Handler
0x0012    rjmp       TIM2_COMP   ; Timer2 Compare Handler
0x0014    rjmp       TIM2_OVF    ; Timer2 Overflow Handler
0x0016    rjmp       TIM1_CAPT   ; Timer1 Capture Handler
0x0018    rjmp       TIM1_COMPA  ; Timer1 CompareA Handler
0x001A    rjmp       TIM1_COMPB  ; Timer1 CompareB Handler
0x001C    rjmp       TIM1_OVF    ; Timer1 Overflow Handler
0x001E    rjmp       TIM0_COMP   ; Timer0 Compare Handler
0x0020    rjmp       TIM0_OVF    ; Timer0 Overflow Handler
0x0022    rjmp       SPI_STC     ; SPI Transfer Complete Handler
0x0024    rjmp       USART0_RXC  ; USART0 RX Complete Handler
0x0026    rjmp       USART0_DRE  ; USART0,UDR Empty Handler
0x0028    rjmp       USART0_TXC  ; USART0 TX Complete Handler
0x002A    rjmp       ADC         ; ADC Conversion Complete Handler
0x002C    rjmp       EE_RDY      ; EEPROM Ready Handler
0x002E    rjmp       ANA_COMP    ; Analog Comparator Handler
0x0030    rjmp       TIM1_COMPC  ; Timer1 CompareC Handler
0x0032    rjmp       TIM3_CAPT   ; Timer3 Capture Handler
0x0034    rjmp       TIM3_COMPA  ; Timer3 CompareA Handler
0x0036    rjmp       TIM3_COMPB  ; Timer3 CompareB Handler
0x0038    rjmp       TIM3_COMPC  ; Timer3 CompareC Handler
0x003A    rjmp       TIM3_OVF    ; Timer3 Overflow Handler
0x003C    rjmp       USART1_RXC  ; USART1 RX Complete Handler
0x003E    rjmp       USART1_DRE  ; USART1,UDR Empty Handler
0x0040    rjmp       USART1_TXC  ; USART1 TX Complete Handler
0x0042    rjmp       TWI         ; Two-wire Serial Interface Handler
0x0044    rjmp       SPM_RDY     ; SPM Ready Handler
;
0x0046    RESET: ldi r16, high(RAMEND) ; Main program start
0x0047           out SPH,r16           ; Set Stack Pointer
0x0048           ldi r16, low(RAMEND)
0x0049           out SPL,r16
0x004A           sei                   ; Enable interrupts
... ... ... ...
```

Figure 1: Intialization of Interrupt Vectors

You do have to setup all the vectors for every program. Figure 2 shows you a small program that uses external interrupts. Read through the code carefully and try to find out what is happenging, take note of the instructions when entering and before exiting the interrupt mode.

```
.include "m64def.inc"

.def temp =r16

.equ HIGH_LEDS  = 0b11110000
.equ LOW_LEDS   = 0b00001111

        jmp RESET

.org INT0addr
        jmp EXT_INT0
.org INT1addr
        jmp EXT_INT1

RESET:
        ldi temp, low(RAMEND)
        out SPL, temp
        ldi temp, high(RAMEND)
        out SPH, temp

        ser temp
        out DDRC, temp
        clr temp
        out PORTC, temp
        out DDRD, temp
        out PORTD, temp

        ldi temp, (2 << ISC10) | (2 << ISC00)

        sts EICRA, temp

        in temp, EIMSK
        ori temp, (1<<INT0) | (1<<INT1)
        out EIMSK, temp

        sei
        jmp main
```

continues on the next page...

```
EXT_INT0:
        push temp
        in   temp, SREG
        push temp

        ldi temp, HIGH_LEDS
        out PORTC, temp

        pop temp
        out SREG, temp
        pop temp
        reti

EXT_INT1:
        push temp
        in temp, SREG
        push temp

        ldi temp, LOW_LEDS
        out PORTC, temp

        pop temp
        out SREG, temp
        pop temp
        reti

; main - does nothing but increment a counter
main:
        clr temp
loop:
        inc temp
        rjmp loop
```

Figure 2: interrupt.asm

Use the patch cables and connect PB0 to PD0 and PB1 to PD1, also connect PC0-PC7 to LED0-LED7. Assemble and download the program onto the AVR Microcontroller Board. The program can be invoked by press the PB0 and PB1 push button at the bottom right corner of the board. Be ready to explain your observation to the Laboratory assessor.

4.4.1 Watchdog Timer

The Watchdog Timer (WDT) runs independent of the rest of the system, causing system resets whenever it times out. However, the application software should ensure that the timeout never occurs by resetting the WDT periodically as long as the software is in a known healthy state. If the system hangs or program execution is corrupted, the WDT will not receive its periodic reset,

and will eventually time out and cause a system reset.

The WDT in all new AVR devices also has the ability to generate interrupts instead of resetting the device. Since the WDT runs from its own independent clock, it can be used to wake up the AVR from all sleep modes. This makes it an ideal wakeup timer, easily combined with ordinary operation as a system reset source. The interrupt can also be used to get an early warning of a upcoming Watchdog System Reset, so that vital parameters can be backed up to non-volatile memory.

When the Watchdog Timer (WDT) period has expired, a WDT timeout occurs. The timeout period is adjusted using a configurable prescaler, which divides the WDT oscillator clock by a constant factor. Executing the WDR (Watchdog Reset) instruction resets the timer value. The application software using the WDT must be designed so that it executes the WDR instruction periodically whenever it decides that the system still operates correctly. The timer value is automatically reset on system reset and when disabling the WDT.

When using the Watchdog Timer it is important to know that if the Watchdog Always On (WDTON) fuse is programmed, the only possible operation mode is WDT System Reset Mode. This security feature prevents software from enabling the WDT Interrupt Mode unintentionally, which could disable the WDT System Reset functionality. When the WDTON fuse is unprogrammed, the WDT Interrupt Mode can be used.

As mentioned above, the WDT is independent from the rest of the system. It has its own internal oscillator, which runs as long as one of the WDT operating modes is enabled. This ensures safe operation even if the main CPU oscillator fails. Even if the software designers never intended to use the WDT, it could be enabled unintentionally, e.g. by a runaway pointer or brown-out condition. Therefore the startup code should always check the Reset Flags and take appropriate action if a WDT System Reset has occurred, even if the application does not use the WDT. The various settings and functions can be combined to use the WDT for different purposes.

Write a program using AVR assembly language that enables the watchdog timer and resets it in your main program before it generates a RESET interrupt. After 5 seconds has passed, your program stops resetting the watchdog timer. So the watchdog timer will generate a RESET interrupt. You program should turn all LEDs on when the watchdog timer generates a RESET interrupt. To know if 5 seconds has passed, you need to use software delay, i.e. executing instructions, instead of using a timer. Read ATMega64 Data Sheet (Pages 54-56) for the details about the Watchdog Timer. Assemble your program using AVR Studio, and run it on the AVR. Show your working program to the lab assessor. Remember to save and delete your work before you leave the laboratory.

Checkpoint 1:	Signature:

Write a program using AVR assembly language that turns on and off every 1 second by using the Timer 0 Interrupt. Your program should enable the watchdog timer and periodically reset it before it generates a RESET signal. Read Mega64 Data Sheet for the details about Timer 0. Assemble your program using AVR Studio, and run it on the AVR Microcontroller Board. Show your working program to the Laboratory assessor. Remember to save and delete your work before you leave the laboratory.

Checkpoint 2:	Signature:

4.5 Part 2: I/O Interfacing

4.5.1 Keypad

Many applications require driving LEDs along with an interface to a keypad. Implementing such designs usually involves using up significant amounts of the processors I/O lines. The 4x4 Key Matrix is can be connected to any of the ports of ATmega64. The keypad sampling is as follows:

1. The columns are connected to output pins, and the rows are connected to input pins.
2. Each column is sequentially driven to a low voltage while at the same instance the four rows are sampled. Since the rows are all held high with interal pull-up resistors of AVR, all four inputs will normally be high. If a key is pressed in a column which is at a low level, that low level will be conducted to the input pin through the closed key and the corresponding row will be sensed as a low.
3. Before a new column is brought low, care should be taken to discharge the input pins.

In other words, to scan a row, one column output is taken low, and then a row of 4 keys is read. The row "data" byte (nibble) that has been read is then compared to the 4 values it would have been had if any of the 4 keys in that row had been pressed. If it matches then that key must have been pressed. The other rows are then processed similiarly taking each of the 3 outputs low in turn. (Note: this is a simplistic scanning system and does not allow two keys to be pressed at any one time).

Figure 3 shows the connection of the 4×4 Matrix Keypad:

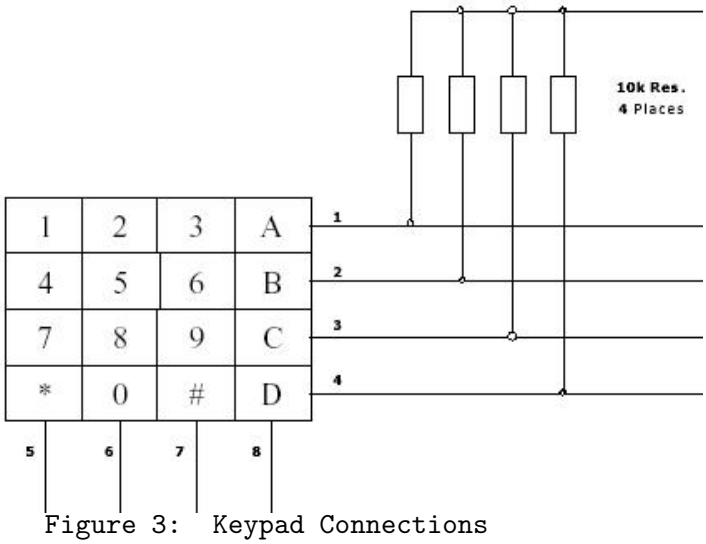

Figure 3: Keypad Connections

Figure 4 shows the AVR assembly language code for the keypad.

```
.include "m64def.inc"
        rjmp RESET

.def temp   =r16
.def row    =r17
.def col    =r18
.def mask   =r19
.def temp2  =r20

.equ PORTDDIR    = 0xF0
.equ INITCOLMASK = 0xEF
.equ INITROWMASK = 0x01
.equ ROWMASK     = 0x0F

RESET:
        ldi     temp, low(RAMEND)
        out     SPL, temp
        ldi     temp, high(RAMEND)
        out     SPH, temp

        ldi temp, PORTDDIR      ; columns are outputs, rows are inputs
        out     DDRD, temp
        ser temp
        out DDRC, temp          ; Make PORTC all outputs
        out PORTC, temp         ; Turn on all the LEDs

; main - Loops while scanning the keypad to find which key is pressed.
main:
        ldi mask, INITCOLMASK   ; initial column mask
        clr     col             ; initial column
colloop:
        out     PORTD, mask     ; set column to mask value
                                ; (sets column 0 off)

        ldi temp, 0xFF          ; implement a delay so the
                                ; hardware can stabilize
delay:
        dec temp
        brne delay
        in temp, PIND           ; read PORTD
        andi temp, ROWMASK      ; read only the row bits
        cpi temp, 0xF           ; check if any rows are grounded
```

continues on the next page...

```
        breq nextcol            ; if not go to the next column
        ldi mask, INITROWMASK   ; initialise row check
        clr     row             ; initial row
rowloop:
        mov temp2, temp
        and temp2, mask         ; check masked bit
        brne skipconv           ; if the result is non-zero,
                                ; we need to look again
        rcall convert           ; if bit is clear, convert the bitcode
        jmp main                ; and start again

skipconv:
        inc row                 ; else move to the next row
        lsl mask                ; shift the mask to the next bit
        jmp rowloop
nextcol:
        cpi col, 3              ; check if we're on the last column
        breq main               ; if so, no buttons were pushed,
                                ; so start again.

        sec                     ; else shift the column mask:
                                ; We must set the carry bit
        rol mask                ; and then rotate left by a bit,
                                ; shifting the carry into
                                ; bit zero.  We need this to make
                                ; sure all the rows have
                                ; pull-up resistors

        inc col                 ; increment column value
        jmp colloop             ; and check the next column


; convert function - Converts the row and column given to a binary
; number and also outputs the value to PORTC.
; Inputs come from registers row and col and output is in temp.

convert:
        cpi col, 3              ; if column is 3 we have a letter
        breq letters
        cpi row, 3              ; if row is 3 we have a symbol or 0
        breq symbols
```

continues on the next page...

```
        mov temp, row           ; otherwise we have a number (1-9)
        lsl temp                ; temp = row * 2
        add temp, row           ; temp = row * 3
        add temp, col           ; add the column address
                                ; to get the offset from 1
        inc temp                ; add 1.  Value of switch is
                                ; row*3 + col + 1.
        jmp convert_end

letters:
        ldi temp, 0xA
        add temp, row           ; increment from 0xA by the row value
        jmp convert_end
symbols:
        cpi col, 0              ; check if we have a star
        breq star
        cpi col, 1              ; or if we have zero
        breq zero
        ldi temp, 0xF           ; we'll output 0xF for hash
        jmp convert_end
star:
        ldi temp, 0xE           ; we'll output 0xE for star
        jmp convert_end

zero:
        clr temp                ; set to zero

convert_end:
        out PORTC, temp         ; write value to PORTC
        ret                     ; return to caller
```

Figure 4: keypad.asm

4.5.2 Liquid Crystal Display

The AVR Microntroller Board comes with a 2×16 character Liquid Crystal Display (LCD) module. This module can be controlled via any of the ATmega64 ports. Below are the pin descriptions.

Signal Name	No. of Lines	Input/Output	Function
DB4 - DB7	4	Input/Output	4 lines of high order data bus. Bi-directional transfer of data between MPU and module is done through these lines. Also DB7 can be used as a busy flag. These lines are used as data in 4 bit operation.
DB0 - DB3	4	Input/Output	4 lines of low order data bus. Bi-directional transfer of data between MPU and module is done through these lines. In 4 bit operation, these are not used and should be grounded.
BE	1	Input	Enable - Operation start signal for data read/write.
RS	1	Input	Register Select "0": Instruction register (Write) : Busy flag; Address counter (Read) "1": Data register (Write, Read)
R/W	1	Input	Signal to select Read or Write "0": Write "1": Read
BL	1	Input	Back light
DS	1	Input	Data bus buffer enable

In order to write to the LCD, here is the list of things you must satisfy.

Busy Flag

When the busy flag is high or "1" the module is performing an internal operation and the next instruction will not be accepted. The busy flag outputs to DB7 when RS=0 and a read operation is performed. The next instruction must not be written until ensuring that the busy flag is low or "0".

Initialization

The display can be initialized using the internal reset circuit if the Internal Power Supply Reset timing below is met.

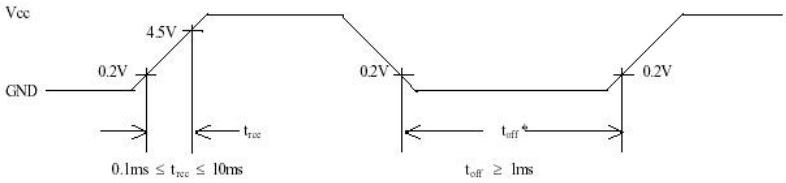

Figure 5: Initialization

Note: t_{off} represents the time of power off condition for a momentary power supply dip or when cycling power off then on.

If the above conditions are met, the busy flag will go active 10ms after Vcc rises to 4.5V.

Selection of Registers

RS	R/W	Operation
0	0	IR write, internal operation (display clear etc)
0	1	Busy flag (DB7), Address counter (DB0-DB6) read
1	0	DR write, Internal Operation (DR DD RAM or CG RAM)
1	1	DR Read, Internal Operation (DD RAM or CG RAM)

In general, each character display can be operated in either 4 or 8 bit mode. The next two figures shows you the steps you must follow to complete the initialization phase. Note: appropriate timing is very important, when you doing this in software, you have to setup appropriate delay function to meet the timing constraint.

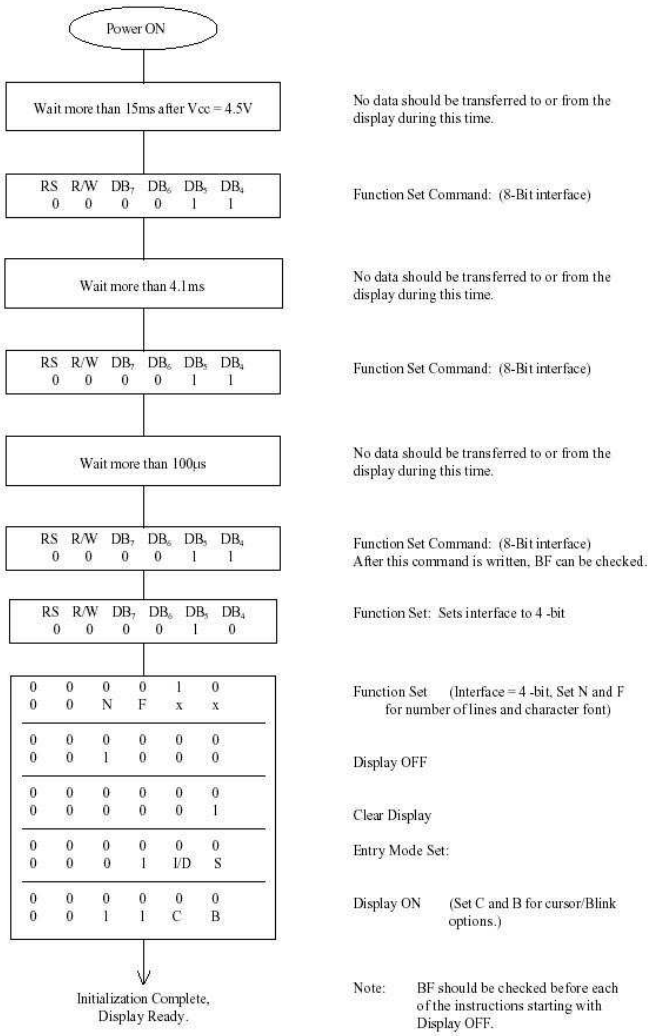
Power ON
Wait more than 15ms after Vcc = 4.5V
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4
0 0 0 0 1 1
Function Set Command: (8-Bit interface)
Wait more than 4.1ms
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4
0 0 0 0 1 1
Function Set Command: (8-Bit interface)
Wait more than 100µs
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4
0 0 0 0 1 1
Function Set Command: (8-Bit interface)
After this command is written, BF can be checked.
RS R/W DB7 DB6 DB5 DB4
0 0 0 0 1 0
Function Set: Sets interface to 4 -bit
0 0 0 0 1 0
0 0 N F x x
Function Set (Interface = 4 -bit, Set N and F for number of lines and character font)
0 0 0 0 0 0
0 0 1 0 0 0
Display OFF
0 0 0 0 0 0
0 0 0 0 0 1
Clear Display
0 0 0 0 0 0
0 0 0 1 I/D S
Entry Mode Set:
0 0 0 0 0 0
0 0 1 1 C B
Display ON (Set C and B for cursor/Blink options.)
Initialization Complete,
Display Ready.
Note: BF should be checked before each of the instructions starting with Display OFF.

Figure 6: 4-bit Initialization

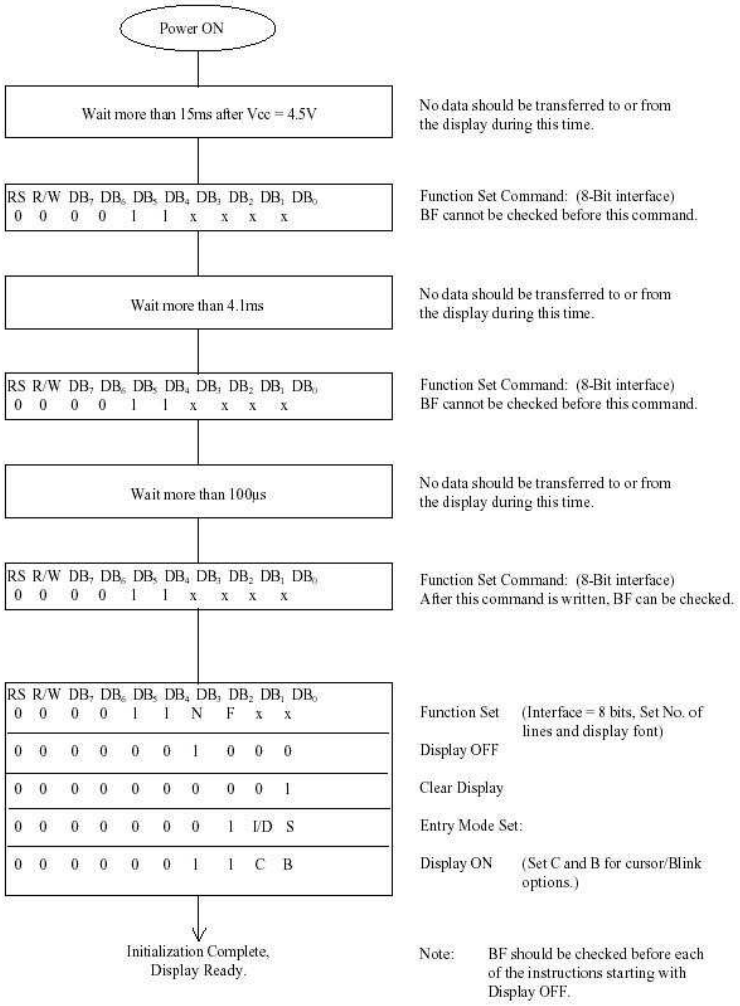
Power ON
Wait more than 15ms after Vcc = 4.5V
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 x x x x
Function Set Command: (8-Bit interface)
BF cannot be checked before this command.
Wait more than 4.1ms
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 x x x x
Function Set Command: (8-Bit interface)
BF cannot be checked before this command.
Wait more than 100µs
No data should be transferred to or from the display during this time.
RS R/W DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 x x x x
Function Set Command: (8-Bit interface)
After this command is written, BF can be checked.
RS R/W DB7 DB6 DB5 DB4 DB3 DB2 DB1 DB0
0 0 0 0 1 1 N F x x
Function Set (Interface = 8 bits, Set No. of lines and display font)
0 0 0 0 0 0 1 0 0 0
Display OFF
0 0 0 0 0 0 0 0 0 1
Clear Display
0 0 0 0 0 0 0 1 I/D S
Entry Mode Set:
0 0 0 0 0 0 1 1 C B
Display ON (Set C and B for cursor/Blink options.)
Initialization Complete, Display Ready.
Note: BF should be checked before each of the instructions starting with Display OFF.

Figure 7: 8-bit Initialization

Instructions/Data are written to the display using the signal timing characteristics found in Figure 8 or Figure 9.

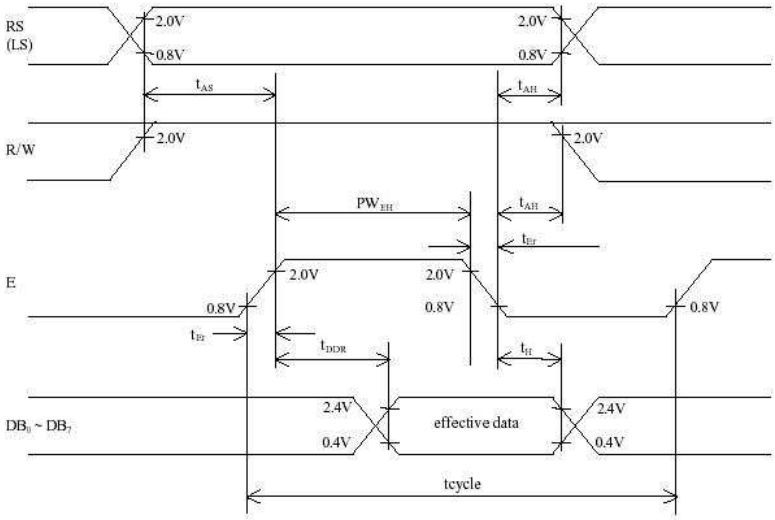

Figure 8: Read from LCD Module Timing Diagram

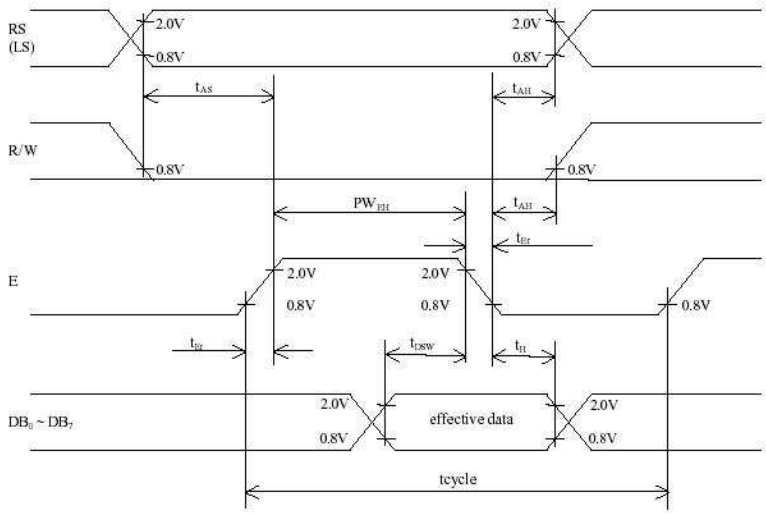

Figure 9: Write to LCD Module Timing Diagram

Control and programming the LCD Module correctly and efficiently can be quite complicated and is beyond the scope of this document. You will need to consult the **Optrex Character LCD Module User's Manual** for more information. You can find this document on the course website.

Write an assembler program called *echo.asm* which receives an character typed in from the keypad and prints it on the LCD. When a line is full, start from the beginning of a line. Assemble and run your program using AVR Studio, download the *hex* file and show your working program to the Laboratory assessor. Remember to save and delete your work before you leave the laboratory.

Checkpoint 3:	Signature:

Write an assembler program called *hex2dec.asm* which receive a *hex* number typed in from the keypad and print the corresponding decimal number on the LCD. Assemble and run your program using AVR Studio, download the *hex* file and show your working program to the Laboratory assessor. Remember to save and delete your work before you leave the laboratory.

Checkpoint 4:	Signature:

4.6 Part 3 (Optional)

4.6.1 Watchdog System Monitor

Write a program in AVR assembly language satisfying the following requirements.

1. Enables the watchdog timer at the beginning.
2. Turns all LEDs on and off every 1 second by using the Timer 0 Interrrupt.
3. Displays the time in seconds that has passed on LCD.
4. Stops resetting the watchdog timer after 15 seconds has passed.
5. Turn on all LEDs when a RESET is generated.

Assemble your program using AVR Studio and run it on the AVR Microcontroller Board. Show your working program to the Laboratory assessor. Remember to save and delete your work before you leave the laboratory.

Checkpoint 4:	Signature: